Software Testing

Inhaltsverzeichnis

Ablauf des Seminars

- Motivation des Testens
- Grundbegriffe (kurz)
- Testen im Software-Lebenszyklus
  - Vorgehensmodelle
  - Teststufen (Komponenten-, Integrations-, System-, Abnahmetest)
  - Last- und Stresstest
- Statische Tests
- Dynamische Tests (Schwerpunktthema)
  - Black-Box-Verfahren
  - White-Box-Verfahren

Es werden Schwerpunkte auf praktische Elemente des Testens gelegt. Die theoretischen Grundlagen sind Fleißarbeit in der Eigenverantwortung der Teilnehmer und können in der einschlägigen Literatur (siehe Buchempfehlung) leicht nachgeholt werden.

Motivation zum Testen

Kleine Beispiele aus der Praxis

z.B. unter https://de.wikipedia.org/wiki/Liste_von_Programmfehlerbeispielen gibt es schöne Beispiele aus der Praxis welche fatalen Auswirkungen eine ungenügende Testabdeckung haben können. Zwei bekannte Beispiele sind folgende:

Teurer Fehler bei Float und Integer:

Am 4. Juni 1996 sprengte sich die erste **Ariane-5-Rakete** (Startnummer V88) der Europäischen Raumfahrtbehörde 40 Sekunden nach dem Start in vier Kilometern Höhe automatisch. Der Programmcode für (unter anderem) die Vorstart-Ausrichtung war von der Ariane 4 übernommen worden, lief unnötigerweise auch nach dem Start weiter und funktioniert dabei nur in einem von der Ariane 4 nicht überschreitbaren Bereich der Horizontalgeschwindigkeit. Als dieser Bereich von der Ariane 5 verlassen wurde, da sie höhere Horizontalgeschwindigkeiten erreicht als die Ariane 4, schlug der Fehler auf die Trägheits-Steuersysteme durch und diese schalteten sich weitgehend ab. Bei der Programmierung war es zu einem Fehler bei der Typumwandlung gekommen. Als von Float nach Integer umgewandelt wurde und der **Wert 32 768** erreichte, entstand ein Überlauf. Dieser Überlauf hätte durch die verwendete Programmiersprache Ada eigentlich entdeckt und behandelt werden können. Diese Sicherheitsfunktionalität ließen die Verantwortlichen jedoch abschalten.

Der Schaden betrug etwa 370 Millionen US-Dollar.

Peinliche Division durch 0

Die USS Yorktown, ein weitgehend computerisiertes Schiff der Ticonderoga-Klasse, trieb nach Aussage der Navy bei einem Manöver im September 1997 knapp drei Stunden hilflos im Mittelmeer, nachdem ein Ingenieur "0" direkt in die Datenbank eingegeben hatte um einen fehlerhaften Eintrag eines Sensors zu korrigieren. Die Software verwendete diesen Wert für Divisionen, die Folge waren „Division durch null“-Fehler, welche von der Software nicht korrekt abgefangen wurden. Es füllte sich im weiteren Verlauf der temporäre Speicher des Systems und als dieser voll war, wurde beim Überlauf der angrenzende Speicherbereich, der des Antriebssystems, überschrieben. Das Antriebssystem und das Rechnernetzwerk, das aus mehreren Windows-NT-Servern bestand, fielen aus, da sich der Fehler im Netzwerk ausgebreitet hatte. Es dauerte über drei Stunden, die Server zu reanimieren.

CrowdStrike-Update bricht IT-Infrastruktur (19. Juli 2024)

Ein fehlerhaftes Update der CrowdStrike Falcon-Sensor-Software führte zu einem der größten globalen IT-Ausfälle, den die Welt bis dahin erlebt hatte. Rund 8,5 Millionen Windows-Systeme stürzten ab oder verharrten in Boot-Loops – ein Albtraum-Szenario. functionize.com (siehe https://de.wikipedia.org/wiki/Crowdstrike-Computerausfall_2024)

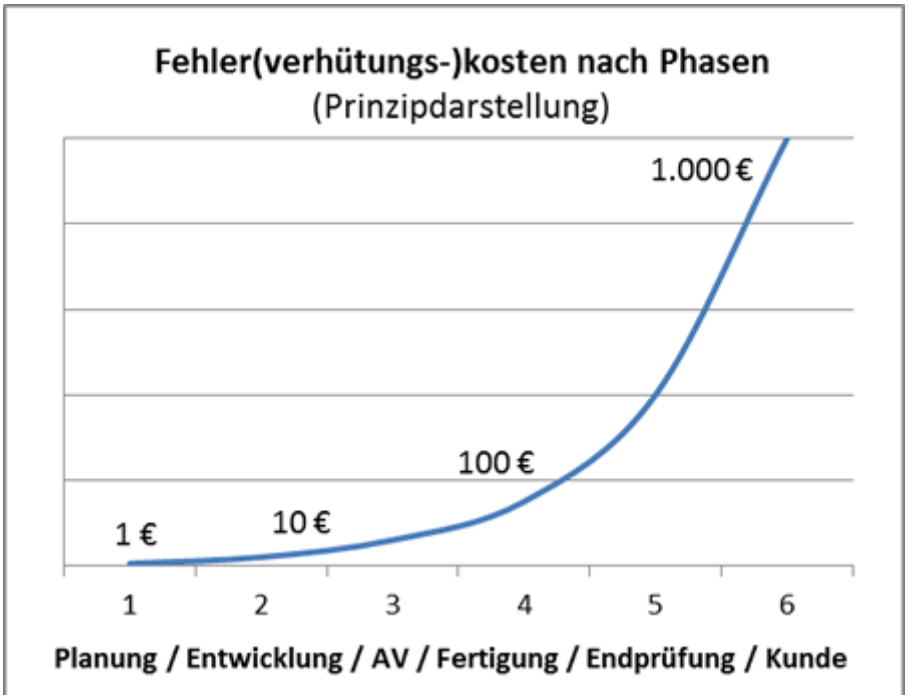
Folgen:

- Flughäfen, Bahnhöfe, Krankenhäuser und Banken weltweit betroffen.
- Delta Air Lines erlitt über 7 000 Flugstreichungen, mit etwa 1,3 Millionen betroffenen Passagieren.
- Die Airline kalkulierte Verluste von rund 550 Millionen USD.
- Delta verklagte CrowdStrike wegen grober Vernachlässigung; das Verfahren läuft noch.

Dieses Szenario ist ein Paradebeispiel dafür, wie zentralisierte Softwarelandschaften und mangelnde Tests das gesamte global vernetzte System ins Wanken bringen können. Eine Art „IT-Monokultur“, in der ein Fehler alle umhaut. techinformed.com

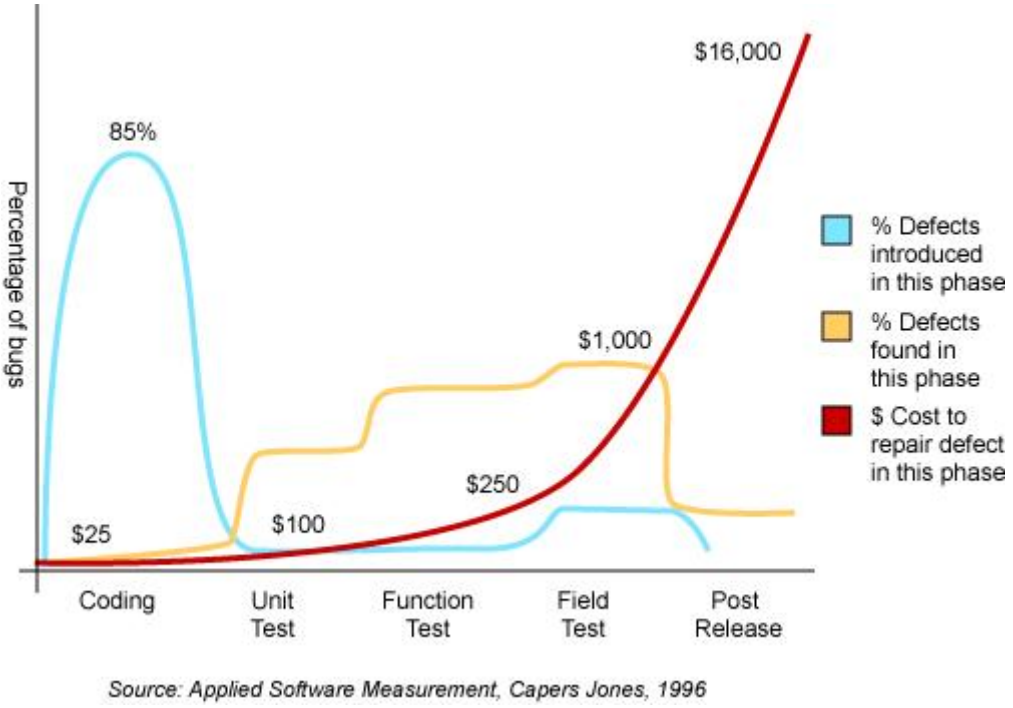
Kosten von Fehlern

Grundsätzlich gilt, dass ein Fehler teurer wird, je später er entdeckt und behoben wird. Einer der Ansätze ist die 10er-Regel:



(Quelle: http://olev.de/0/10er-regl.htm)

Bezeichnend ist der exponentielle Anstieg der Kosten was auch noch einmal schön in diesem Zusammenhang deutlich wird:



(Quelle: https://deanondelivery.com/product-managers-do-you-know-how-much-your-bugs-cost-72b6e36e7684)

Getting Things Done

- [x] Ein ganz wichtiger Punkt in der Softwareentwicklung ist der, an dem man eine Aufgabe als erledigt betrachten kann.
- [ ] Wie **definiert** man nun diesen Punkt?
- [ ] Wann genau ist eine Aufgabe **erledigt**?

Hierbei sind Tests ein sehr hilfreiches Mittel. Mit dem Test können wir definieren was das Ziel ist und wir können immer wieder überprüfen, ob dieses Ziel auch in Zukunft (nach anderen Änderungen) noch erreicht wird.

"Never touch a running system!" ist ein Ausspruch für schlechtes Testen. Wenn ich eine gute und saubere Testabdeckung habe kann ich gerne ändern. Wenn danach alle meine Tests wieder sauber durchlaufen haben meine Änderungen keine Seiteneffekte. Ich bin frei zu ändern.

Praxistipp:

Wenn eine neue Aufgabe begonnen wird, wird hierfür (hoffentlich) von dem **Develop-Branch** ein neuer **Feature-Branch** abgezweigt. Bevor man mit der Aufgabe beginnt, sollte man sicherstellen, dass auf diesem Branch alle Tests funktionieren. Dann ist man sich sicher, dass, wenn nachher irgendwelche Tests nicht funktionieren, man selbst die Ursache war.

Vorgehen (Empfehlung):

- Feature-Branch erstellen und auschecken
- neuen Branch testen

- Änderungen machen
- Branch testen
- Pull Request stellen
- TDD, wenn möglich.

Broken-Window-Theorie

Wenn man einmal schludrig mit den Tests ist und die Qualität vernachlässigt, wird die Vernachlässigung immer schlimmer.

- Kein Vertrauen in die Software und die eigene Arbeit
- Keine Wertschätzung in das Produkt

Grundbegriffe

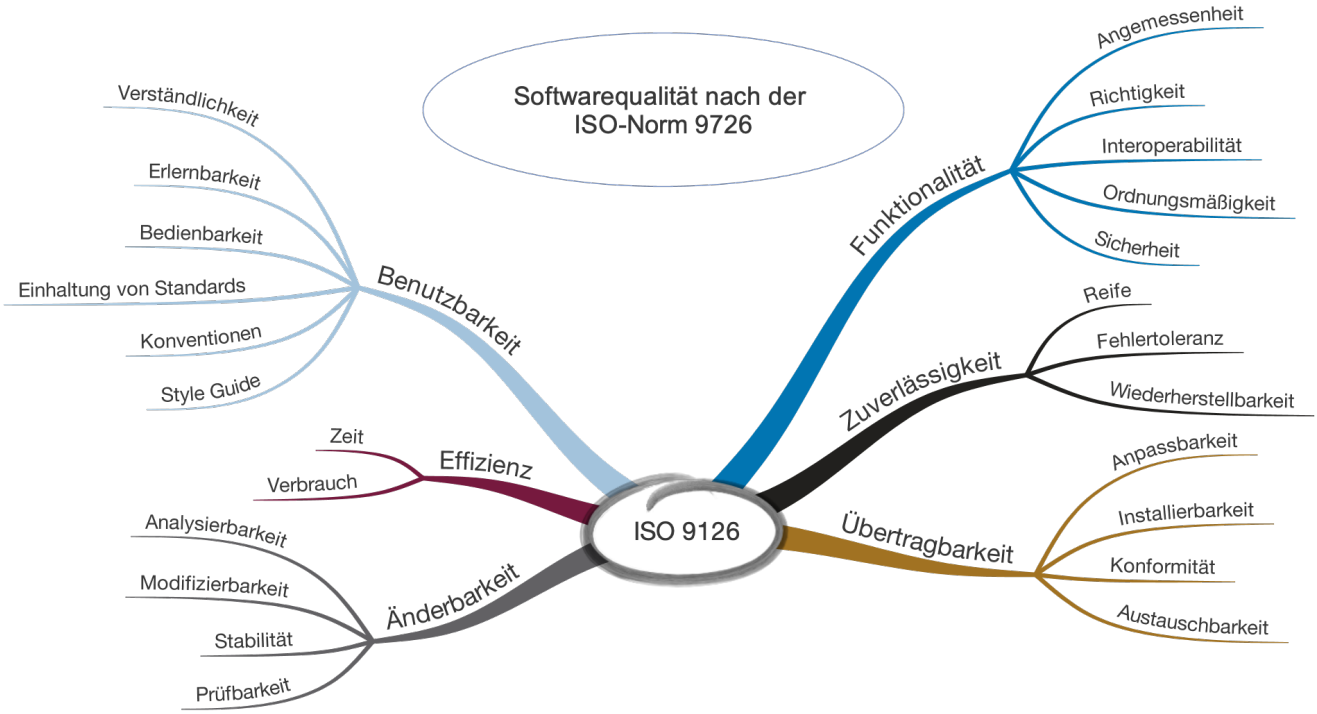
Grundbegriff Fehler

- Fehlerbegriff
  - Nichterfüllung festgelegter Anforderungen
  - Abweichung von Ist- und Soll-Verhalten
- Fehlerwirkung
  - Ursprung in Fehlerzustand
- Fehlermaskierung
- Fehlhandlung
  - Programmierfehler
  - Vergessene Anforderung

Grundbegriff Test

- Testbegriff
  - Debugging
  - Test → stichprobenhafte Prüfung
- Ziele des Testens
  - Fehlerwirkung nachweisen
  - Qualität bestimmen
  - Vertrauen in das Programm erhöhen
  - Analyse des Programms oder der Dokumentation zur Vorbeugung von Fehlerwirkungen

Grundbegriff Softwarequalität



Grundbegriff Testaufwand

- Testaufwand
  - Fehlerfreiheit ist nicht nachweisbar.
  - Vollständiger Test ist nicht durchführbar, da es zu viele kombinierbare Möglichkeiten gibt.
- Einflussfaktoren für die Bemessung des Testaufwandes
  - Durch Fehler verursachte Kosten können in die Millionen gehen
  - Risiko bestimmt sich aus Eintrittswahrscheinlichkeit und zu erwartender Schadenshöhe
  - Es kann Gefahr für Leib oder gar Leben bestehen
  - Testaufwand in Abhängigkeit von Risiko und Gefährdungsbewertung
  - Kombination aus unterschiedlichen Testverfahren notwendig
  - Testfall-Explosion

Aufgabe

Wie wird wohl der Testaufwand bei folgenden Projekten aussehen?

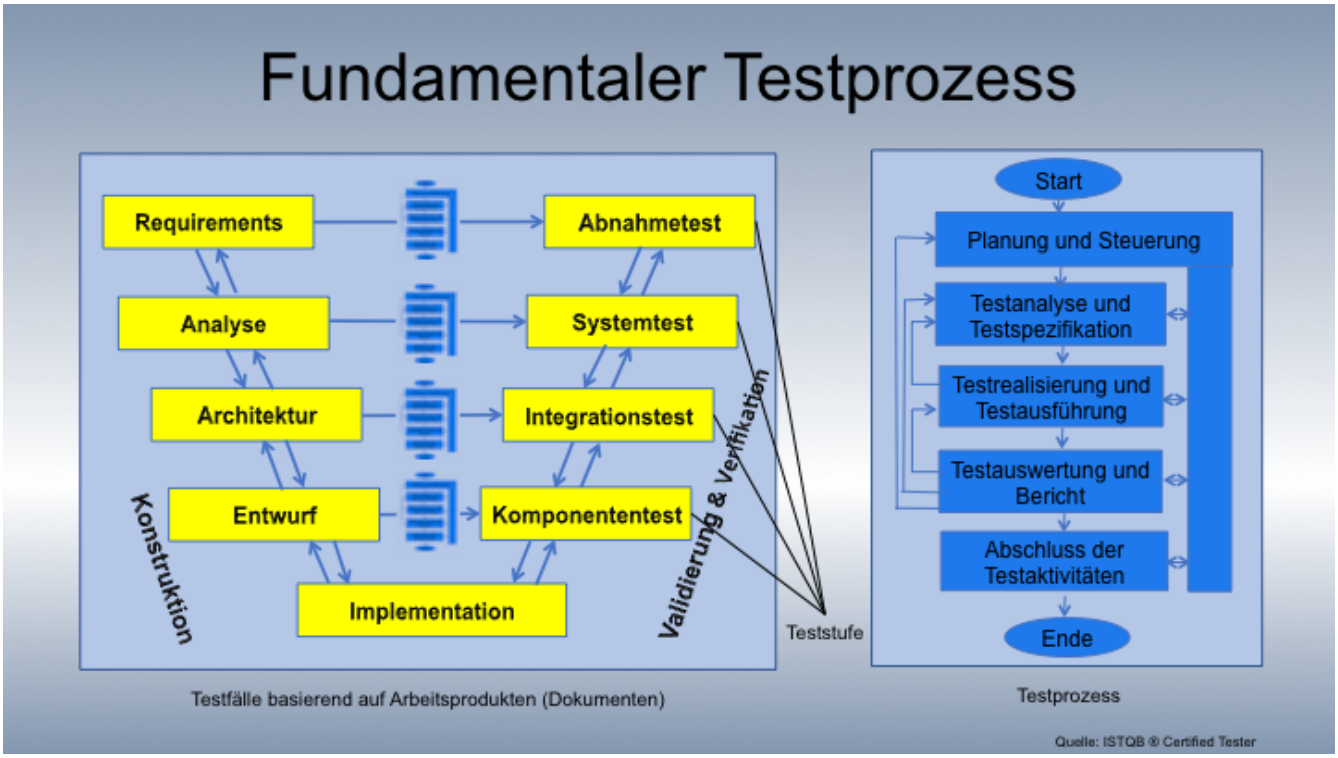
- ▢ Banking-App der BNP Paribas
- ▢ Parkplatz-Sharing für Mitarbeiter eines Unternehmens
- ▢ Steuerungssoftware für Langstreckenraketen
- ▢ Steuerungssoftware für Beatmungssysteme im medizinischen Bereich

In welchen Teilbereichen der Software wird der Schwerpunkt liegen? Wie bewertet ihr das Risiko (wo liegt das Risiko) und wo liegen die Prioritäten im Testen?

Testen im Software-Lebenszyklus

Testplanung und Steuerung

Nach dem ISTQB-Standard stellt sich die Testplanung und Steuerung folgendermaßen dar:



Die ISTQB arbeitet für eine standardisierte Qualifikation von Software-Testern und fungiert als Zertifizierungsinstanz.

Testziele

- Test auf Funktionalität
- Test auf Robustheit
- Test auf Effizienz
- Test auf Wartbarkeit
- Test auf unvollständige oder widersprüchliche Anforderungen

⇒ Ziel muss ein Continuous Deployment sein (= Continuous Delivery + Installation)

Teststufen

- Komponententest
  - JUnit
  - Kleinste Einheit
- Integrationstest

    - Zusammenarbeit von Komponenten
    - auch für Fremdsysteme
    - Top-Down, Bottom-Up, Ad-hoc, Backbone
- Systemtest
    - aus der Sicht des Kunden
    - Prüfung ob Dokumentation und Anforderung stimmt
    - Produktionsumgebungsnahe Testumgebung
- Abnahmetest
    - Ist der Vertrag erfüllt? (vertragliche Akzeptanz)
    - Kann und will der Benutzer damit arbeiten? (Benutzerakzeptanz)
    - Feldtest

Diese Teststufen sollte man **nicht** in einem Wasserfall-Modell sehen und nicht **nacheinander** abarbeiten. Der Umfang der einzelnen Teststufen wächst mit dem Programm. Je früher das Programm in Feldversuchen getestet wird umso eher fallen diesbezügliche Probleme auf.

⇒ Agile Softwareentwicklung!

Aufgabe

Analyse und Erweiterung der Tests im Beispielprojekt.

- ▢ Welche Tests gehören zu welchen Teststufen?
- ▢ Wie würden Tests auf den fehlenden Teststufen aussehen?

Last- und Stresstest

- https://www.testing-board.com/lasttest-tools-uebersicht/
    - Silk Performer
    - JMeter
- SoapUI für Webservices

Aufgabe

- ▢ Erstellen und Analysieren eines Lasttests

Statische Tests

Bei statischen Testverfahren wird die Software nicht ausgeführt (wie bei den dynamischen Testverfahren) sondern analysiert. Dies kann automatisiert durch Software-Tools (Quelltextanalyse) geschehen oder durch die menschlichen Denk- und Analysefähigkeiten (Reviews).

- Quelltextanalyse
  - https://www.eclemma.org/
  - https://de.wikipedia.org/wiki/Liste_von_Werkzeugen_zur_statischen_Codeanalyse
  - https://www.sonarqube.org/
- Review
  - IEEE 1028 (Standard for Software Reviews and Audits)

Dynamische Tests

Black-Box-Verfahren

- Komplettes Programm
- Zugriff "von Außen" ohne Kenntnis des Quelltextes
- Tools z. B.
  - SOAP-UI für WebServices
  - Selenium für GUI-Oberflächen
  - Insomnia für REST-Services

Aufgabe

- ☐ Erstellen eines Selenium-Tests

White-Box-Verfahren

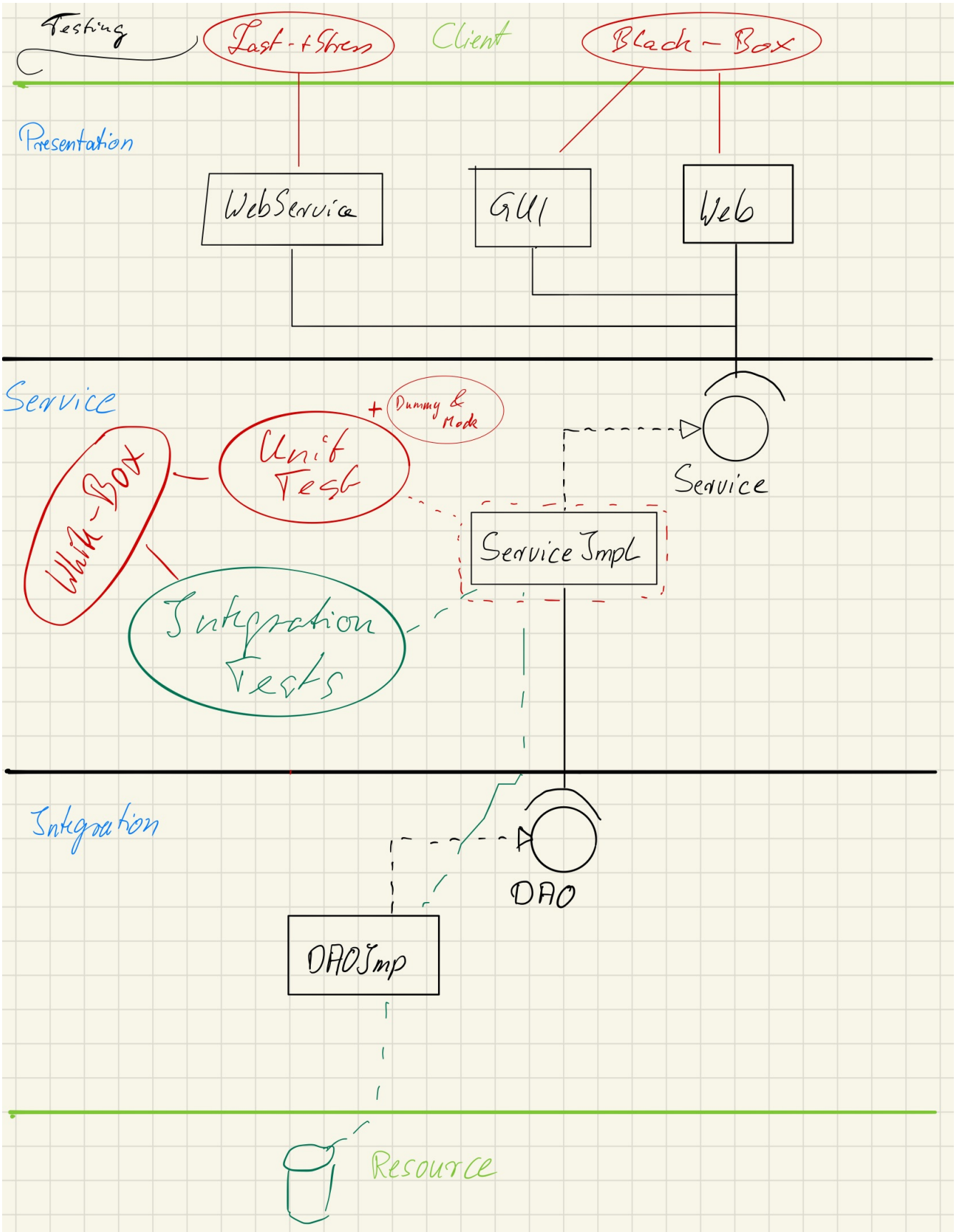
- Quelltext liegt vor
- Überprüfung der Wartbarkeit
- Möglichkeiten des Mockings
- Tools z.B.
  - JUnit-Tests
  - Datenbank-Tests
  - Mocks mit EasyMock und Dummies

Mocks und Dummys

Die Literatur unterscheidet einige Typen von Test-Doubles, sprich Objekten die als Ersatz der eigentlichen Objekte bei Tests eingesetzt werden um die Komplexität des Tests zu reduzieren.

Die gebräuchlichsten Typen sind Dummys und Mocks. Dummys sind sehr primitiv und geben immer den gleichen Wert zurück. Bei Mocks können Verhaltensweisen aufgezeichnet und am Ende verifiziert werden.

Weitere Typen sind "Fake-Objekt", "Test-Stub" oder auch "Test-Spy".


Testing
Load + Stress
Client
Black - Box
Presentation
WebService
GUI
Web
Service
+
Dummy & Mock
Unit Test
White - Box
Service
Service Impl
Integration Tests
Integration
DAO
DAOImp
Resource

Test-Driven-Development (TDD)

Regeln für TDD:

1. Produktivcode darf ohne fehlschlagende Tests nicht geschrieben werden.
2. Es darf nicht mehr Testcode als unbedingt nötig geschrieben werden, um einen Fehler anzuzeigen.
3. Es darf nicht mehr Produktivcode als unbedingt notwendig geschrieben werden, nur damit der vormals fehlgeschlagene Test erfolgreich ist.
4. Tests müssen sauber sein.

Seine Eigenschaften fasst das F.I.R.S.T.-Prinzip zusammen:

- Schnell ("fast"): Tests sollten schnell sein, sodass sie möglichst oft ausgeführt werden können. Tests, die viel Zeit in Anspruch nehmen, werden seltener ausgeführt. Fehler werden dann nicht sofort erkannt.
- Unabhängig ("independent"): Falls Tests voneinander abhängig sind, kann ein Fehler viele weitere anhängige Tests zum Scheitern bringen. Die Fehlersuche ist dann schwer. Unabhängige Tests können in beliebiger Reihenfolge und beliebig oft ausgeführt werden.
- Wiederholbar ("repeatable"): Tests sollten auf allen unterstützten Umgebungen ausführbar sein. Das heißt, sowohl auf der Entwicklungs-, der Staging- und der Produktionsumgebung funktionieren die Tests.
- Selbstvalidierend ("self-validating"): Das Ergebnis eines Tests sollte eindeutig sein. Entweder der Test war erfolgreich oder nicht. Mit Assertions kann man überprüfen, ob das Ergebnis korrekt ist.
- Zur richtigen Zeit ("timely"): Produktiver Code wird nur geschrieben, wenn es einen Test gibt, der fehlschlägt.

(aus Kai Spichale: Testwissen für Java-Entwickler.)

Weitere Hinweise:

- Red-Green-Cycle
- Tests schreiben, bevor die Anforderung realisiert wird ⇒ Der Test richtet sich dann an der Anforderung aus und nicht an der Implementierung.
- Erhöhung der Testabdeckung
- Code-Katas z. B. https://www.youtube.com/watch?v=9flsVKN4tZM

Aufgabe

- ▢ TDD für Passwortüberprüfung

A. Anhang

A1. Links & Quellen

Zusätzliche Infos

- https://www.istqb.org/
- https://de.wikipedia.org/wiki/Liste_von_Programmfehlerbeispielen
- https://deanondelivery.com/product-managers-do-you-know-how-much-your-bugs-cost-72b6e36e7684

Bücher

- Basiswissen Softwaretest
  Andreas Spillner und Tilo Linz
  dpunkt.verlag
  ISBN 978-3-89864-642-0
- Testwissen für Java-Entwickler
  Kai Spichale
  Verlag: Software + Support
  ISBN-13: 9783868024692

Tools

- JMeter https://jmeter.apache.org
- SOAP-UI https://www.soapui.org/
- Selenium https://www.selenium.dev/
- EasyMock https://easymock.org/
- JUnit https://junit.org/junit5/
- Jenkins https://www.jenkins.io/
- Insomnia https://insomnia.rest/

A2. Stichwortverzeichnis

A3. Abkürzungsverzeichnis

DB

Datenbank

ISTQB

International Software Testing Qualifications Board

TDD

Test-Driven-Development